BOOK 15: DATABASES
Chapter 1: Database Concepts

WHY DATABASES MATTER

You built APIs. They store data in arrays.

When the server restarts, everything disappears.

Databases solve this:
- Data persists across restarts
- Multiple users access same data
- Data stays organized
- Queries find data fast
- Relationships connect data

Without databases, applications forget everything.

WHAT IS A DATABASE

A database is organized storage for data.

Think of a library:
- Books are data
- Shelves organize books
- Catalog helps find books
- Librarian manages access

A database management system (DBMS) is the librarian.

Common DBMS:
- PostgreSQL (SQL)
- MySQL (SQL)
- SQLite (SQL)
- MongoDB (NoSQL)

- Redis (Key-Value)

DATABASE TYPES

Relational databases (SQL):

Data stored in tables with rows and columns.
Tables connect through relationships.
Use SQL language.

Example:
- Users table
- Posts table
- Posts reference users by ID

Strengths:
- Structured data
- Complex queries
- Data integrity
- ACID compliance

Examples: PostgreSQL, MySQL, SQLite

Document databases (NoSQL):

Data stored as documents (like JSON).
Documents grouped in collections.
Flexible schema.

Example:
- Users collection
- Each user is a JSON document
- Documents can have different fields

Strengths:
- Flexible structure

- Easy scaling
- Natural for JavaScript
- Fast for simple queries

Examples: MongoDB, CouchDB

Key-value stores:

Data stored as key-value pairs.
Like a giant JavaScript object.

Example:
- "user:123" -> "{name: 'Alice'}"
- "session:abc" -> "{userId: 123}"

Strengths:
- Extremely fast
- Simple operations
- Great for caching

Examples: Redis, Memcached

Graph databases:

Data stored as nodes and edges.
Optimized for relationships.

Example:
- Users are nodes
- Friendships are edges
- Query: "Friends of friends who like X"

Strengths:
- Complex relationships
- Social networks
- Recommendation engines

Examples: Neo4j, Amazon Neptune

SQL VS NOSQL

SQL (Relational):

Fixed schema - define structure before adding data.
Tables with rows and columns.
Relationships through foreign keys.
Powerful query language.
ACID transactions.

Best for:
- Financial data
- Complex relationships
- Data integrity critical
- Complex queries needed

NoSQL (Document):

Flexible schema - structure can vary.
Collections with documents.
Embedded data or references.
Simple query syntax.
BASE transactions (eventually consistent).

Best for:
- Rapid development
- Changing requirements
- Large scale
- Simple queries

ACID PROPERTIES

Relational databases guarantee ACID:

Atomicity:
All operations succeed or all fail.
No partial updates.

```
let transfer = beginTransaction()
transfer.debit(accountA, 100)
transfer.credit(accountB, 100)
transfer.commit()
```

If credit fails, debit is rolled back.

Consistency:
Data always valid according to rules.
Constraints enforced.

Cannot have negative balance if constrained.
Cannot reference non-existent user.

Isolation:
Concurrent transactions don't interfere.
Each sees consistent state.

Two transfers happening simultaneously both work correctly.

Durability:
Committed data survives crashes.
Written to disk, not just memory.

Server crashes after commit, data still there.

DATABASE TERMINOLOGY

Table / Collection:
Group of related records.
Users table, posts collection.

Row / Document:
Single record.
One user, one post.

Column / Field:
Single piece of data.
Name, email, age.

Primary Key:
Unique identifier for each row.
Usually id or _id.

Foreign Key:
Reference to another table.
userId in posts table references users table.

Index:
Speed up queries on specific columns.
Like a book index.

Query:
Request for data.
"Find all users named Alice"

Schema:
Structure definition.
What columns exist, what types.

CHOOSING A DATABASE

Consider these factors:

Data structure:

- Fixed, predictable -> SQL
- Flexible, evolving -> NoSQL

Relationships:
- Many complex relationships -> SQL
- Few or embedded -> NoSQL

Scale:
- Vertical (bigger server) -> SQL
- Horizontal (more servers) -> NoSQL

Consistency:
- Strong consistency needed -> SQL
- Eventual consistency okay -> NoSQL

Team experience:
- SQL knowledge -> PostgreSQL/MySQL
- JavaScript focus -> MongoDB

Common choices:

Web applications:
PostgreSQL or MySQL for main data.
Redis for caching and sessions.

Startups / MVPs:
MongoDB for flexibility.
Easy to change as requirements evolve.

Enterprise:
PostgreSQL for reliability.
Strong ACID compliance.

Real-time applications:
MongoDB or PostgreSQL with good indexing.
Redis for pub/sub.

DATABASE ARCHITECTURE

Client-server model:

Your application (client) connects to database (server).

Application -> Driver -> Database Server -> Storage

Driver:
Library that speaks database protocol.
pg for PostgreSQL, mongodb for MongoDB.

Connection:
Network connection to database.
Usually TCP on specific port.

Connection pool:
Reuse connections for performance.
Don't open/close for every query.

Local vs remote:

Development:
Database runs on same machine.
localhost:5432 for PostgreSQL.
localhost:27017 for MongoDB.

Production:
Database on separate server.
Or managed service (AWS RDS, MongoDB Atlas).

PRACTICE

Research PostgreSQL and MongoDB.

Install one locally.

Understand when to use SQL vs NoSQL.

Think about data for your apps.

WHAT YOU LEARNED

You learned why databases matter.
You learned database types.
You learned SQL vs NoSQL tradeoffs.
You learned ACID properties.
You learned terminology.
You learned how to choose.

The next chapter teaches SQL fundamentals.
BOOK 15: DATABASES
Chapter 2: SQL Fundamentals

WHAT IS SQL

SQL stands for Structured Query Language.

Pronounced "sequel" or "S-Q-L".

SQL communicates with relational databases:
- Create tables
- Insert data
- Query data
- Update data
- Delete data

SQL is declarative. You say WHAT you want, not HOW to get it.

TABLES AND SCHEMAS

Tables hold data in rows and columns.

```
CREATE TABLE users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100) NOT NULL,
 email VARCHAR(255) UNIQUE NOT NULL,
 age INTEGER,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Breaking down:

CREATE TABLE users - Create a table named users

id SERIAL PRIMARY KEY:
- id is the column name
- SERIAL auto-increments (1, 2, 3...)
- PRIMARY KEY makes it the unique identifier

name VARCHAR(100) NOT NULL:
- name is column name
- VARCHAR(100) is text up to 100 characters
- NOT NULL means required

email VARCHAR(255) UNIQUE NOT NULL:
- UNIQUE means no duplicates allowed

age INTEGER:
- INTEGER is whole number
- No NOT NULL means optional (can be NULL)

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP:
- TIMESTAMP stores date and time
- DEFAULT sets value if not provided

DATA TYPES

Common SQL data types:

Text:
VARCHAR(n) - Variable length text up to n characters
TEXT - Unlimited text
CHAR(n) - Fixed length text

Numbers:
INTEGER - Whole numbers
BIGINT - Large whole numbers
DECIMAL(p,s) - Exact decimals (money)
FLOAT - Approximate decimals

Boolean:
BOOLEAN - true or false

Date and time:
DATE - Just date
TIME - Just time
TIMESTAMP - Date and time

Special:
SERIAL - Auto-incrementing integer
UUID - Unique identifier
JSON - JSON data
ARRAY - Array of values

INSERTING DATA

INSERT adds rows:

```
INSERT INTO users (name, email, age)
VALUES ('Alice', 'alice@example.com', 25);
```

Breaking down:

```
INSERT INTO users - Insert into users table
(name, email, age) - Columns to fill
VALUES ('Alice', 'alice@example.com', 25) - Values for those
columns
```

id and created_at auto-fill because of SERIAL and DEFAULT.

Insert multiple rows:

```
INSERT INTO users (name, email, age) VALUES
 ('Alice', 'alice@example.com', 25),
 ('Bob', 'bob@example.com', 30),
 ('Charlie', 'charlie@example.com', 35);
```

QUERYING DATA

SELECT retrieves data:

Get all columns:

```
SELECT * FROM users;
```

Returns all rows with all columns.

Get specific columns:

```
SELECT name, email FROM users;
```

Returns only name and email.

Filter with WHERE:

SELECT * FROM users WHERE age > 25;

SELECT * FROM users WHERE name = 'Alice';

SELECT * FROM users WHERE email LIKE '%@gmail.com';

WHERE filters rows.
LIKE does pattern matching (% is wildcard).

Multiple conditions:

SELECT * FROM users WHERE age > 25 AND name LIKE 'A%';

SELECT * FROM users WHERE age < 20 OR age > 60;

AND requires both conditions.
OR requires either condition.

Sort results:

SELECT * FROM users ORDER BY name;

SELECT * FROM users ORDER BY age DESC;

SELECT * FROM users ORDER BY created_at DESC, name ASC;

ORDER BY sorts results.
ASC ascending (default).
DESC descending.

Limit results:

SELECT * FROM users LIMIT 10;

SELECT * FROM users LIMIT 10 OFFSET 20;

LIMIT returns only n rows.
OFFSET skips n rows (for pagination).

UPDATING DATA

UPDATE modifies existing rows:

UPDATE users SET age = 26 WHERE name = 'Alice';

Breaking down:

UPDATE users - Update users table
SET age = 26 - Set age to 26
WHERE name = 'Alice' - Only rows where name is Alice

WARNING: Without WHERE, updates ALL rows!

UPDATE users SET age = 30;

This sets everyone's age to 30. Be careful!

Update multiple columns:

UPDATE users SET name = 'Alicia', age = 27 WHERE id = 1;

DELETING DATA

DELETE removes rows:

DELETE FROM users WHERE id = 5;

Breaking down:

DELETE FROM users - Delete from users table
WHERE id = 5 - Only row with id 5

WARNING: Without WHERE, deletes ALL rows!

DELETE FROM users;

This deletes everyone. Be careful!

Delete with conditions:

DELETE FROM users WHERE age < 18;

DELETE FROM users WHERE created_at < '2020-01-01';

AGGREGATE FUNCTIONS

Functions that work on multiple rows:

COUNT - Number of rows:

SELECT COUNT(*) FROM users;

SELECT COUNT(*) FROM users WHERE age > 25;

SUM - Add up values:

SELECT SUM(age) FROM users;

AVG - Average:

SELECT AVG(age) FROM users;

MIN and MAX:

SELECT MIN(age), MAX(age) FROM users;

GROUP BY:

SELECT age, COUNT(*) FROM users GROUP BY age;

Groups rows by age, counts each group.

HAVING filters groups:

SELECT age, COUNT(*) FROM users
GROUP BY age
HAVING COUNT(*) > 5;

Like WHERE but for groups.

ALIASES

Rename columns in results:

SELECT name AS username, email AS contact FROM users;

SELECT COUNT(*) AS total_users FROM users;

AS gives column a different name in results.

NULL HANDLING

NULL means no value.

Check for NULL:

SELECT * FROM users WHERE age IS NULL;

SELECT * FROM users WHERE age IS NOT NULL;

Don't use = NULL, use IS NULL.

COALESCE provides default:

SELECT name, COALESCE(age, 0) AS age FROM users;

Returns 0 if age is NULL.

OPERATORS

Comparison:
= Equal
<> or != Not equal
< Less than
> Greater than
<= Less than or equal
>= Greater than or equal

Logical:
AND Both must be true
OR Either must be true
NOT Negates condition

Pattern matching:
LIKE Pattern match (% any chars, _ one char)
ILIKE Case-insensitive LIKE (PostgreSQL)

Range:

BETWEEN x AND y - Value in range
IN (a, b, c) - Value in list

SELECT * FROM users WHERE age BETWEEN 20 AND 30;

SELECT * FROM users WHERE name IN ('Alice', 'Bob', 'Charlie');

PRACTICE QUERIES

Create a posts table:

```
CREATE TABLE posts (
 id SERIAL PRIMARY KEY,
 user_id INTEGER NOT NULL,
 title VARCHAR(200) NOT NULL,
 content TEXT,
 published BOOLEAN DEFAULT false,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Insert posts:

```
INSERT INTO posts (user_id, title, content, published) VALUES
 (1, 'First Post', 'Hello world!', true),
 (1, 'Second Post', 'More content', true),
 (2, 'Draft Post', 'Work in progress', false);
```

Query posts:

```
SELECT * FROM posts WHERE published = true;

SELECT title, created_at FROM posts
WHERE user_id = 1
ORDER BY created_at DESC;

SELECT user_id, COUNT(*) AS post_count
FROM posts
```

```
GROUP BY user_id;
```

PRACTICE

Write CREATE TABLE statements.

Practice INSERT, SELECT, UPDATE, DELETE.

Use WHERE with different conditions.

Try aggregate functions.

Build sample data and query it.

WHAT YOU LEARNED

You learned SQL syntax basics.
You learned data types.
You learned CRUD operations.
You learned filtering and sorting.
You learned aggregate functions.
You learned NULL handling.

The next chapter teaches PostgreSQL with Node.js.

BOOK 15: DATABASES

Chapter 3: PostgreSQL with Node.js

WHY POSTGRESQL

PostgreSQL is a powerful open-source relational database.

Features:
- Full SQL support
- ACID compliance
- JSON support
- Advanced data types
- Excellent performance
- Great documentation

Popular for web applications, APIs, and enterprise software.

INSTALLING POSTGRESQL

On Ubuntu/Debian:

```
sudo apt update
sudo apt install postgresql postgresql-contrib
```

Breaking down:

sudo apt update - Update package list
sudo apt install - Install packages
postgresql - The database server
postgresql-contrib - Additional utilities

Start the service:

```
sudo systemctl start postgresql
sudo systemctl enable postgresql
```

Breaking down:

systemctl start - Start the service now
systemctl enable - Start on boot

On macOS with Homebrew:

```
brew install postgresql
brew services start postgresql
```

Check installation:

```
psql --version
```

POSTGRESQL BASICS

Connect to PostgreSQL:

```
sudo -u postgres psql
```

Breaking down:

```
sudo -u postgres - Run as postgres user
psql - PostgreSQL command line client
```

Create a database:

```
CREATE DATABASE myapp;
```

List databases:

```
\l
```

Connect to database:

```
\c myapp
```

Create a user:

```
CREATE USER myuser WITH PASSWORD 'mypassword';
```

Grant privileges:

GRANT ALL PRIVILEGES ON DATABASE myapp TO myuser;

Quit:

\q

NODE.JS POSTGRESQL DRIVER

Install pg package:

npm install pg

Breaking down:

pg - PostgreSQL client for Node.js

Basic connection:

```
const { Client } = require("pg")

let client = new Client({
 host: "localhost",
 port: 5432,
 database: "myapp",
 user: "myuser",
 password: "mypassword"
})

async function connect() {
 await client.connect()
 console.log("Connected to PostgreSQL")
}
```

connect()

Breaking down:

Client - Single connection to database
host - Database server address
port - PostgreSQL default port 5432
database - Database name
user - Database user
password - User password

RUNNING QUERIES

Simple query:

```
const { Client } = require("pg")

let client = new Client({
 connectionString: "postgresql://
myuser:mypassword@localhost:5432/myapp"
})

async function main() {
 await client.connect()

 let result = await client.query("SELECT NOW()")
 console.log(result.rows[0])

 await client.end()
}

main()
```

Breaking down:

connectionString - All connection info in one string
client.query() - Execute SQL query

result.rows - Array of returned rows
client.end() - Close connection

PARAMETERIZED QUERIES

Never concatenate user input into SQL. Use parameters.

Dangerous (SQL injection):

```
let name = req.body.name
await client.query(`SELECT * FROM users WHERE name =
'${name}'`)
```

If name is: ' OR '1'='1
Query becomes: SELECT * FROM users WHERE name = '' OR
'1'='1'
This returns all users!

Safe (parameterized):

```
let name = req.body.name
await client.query("SELECT * FROM users WHERE name = $1",
[name])
```

$1 is a placeholder.
Second argument is array of values.
Driver safely escapes the value.

Multiple parameters:

```
await client.query(
 "SELECT * FROM users WHERE age > $1 AND name LIKE $2",
 [25, "%Alice%"]
)
```

$1 gets 25, $2 gets "%Alice%".

CONNECTION POOLING

Creating connections is expensive. Pools reuse connections.

```
const { Pool } = require("pg")

let pool = new Pool({
 host: "localhost",
 port: 5432,
 database: "myapp",
 user: "myuser",
 password: "mypassword",
 max: 20
})

async function getUsers() {
 let result = await pool.query("SELECT * FROM users")
 return result.rows
}
```

Breaking down:

Pool - Manages multiple connections
max: 20 - Maximum 20 connections in pool
pool.query() - Gets connection, runs query, releases connection

Pool automatically:
- Creates connections as needed
- Reuses idle connections
- Limits total connections
- Handles connection errors

CRUD OPERATIONS

Create table:

```
async function createTable() {
 await pool.query(`
 CREATE TABLE IF NOT EXISTS users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100) NOT NULL,
 email VARCHAR(255) UNIQUE NOT NULL,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
 )
 `)
}
```

Insert (Create):

```
async function createUser(name, email) {
 let result = await pool.query(
 "INSERT INTO users (name, email) VALUES ($1, $2) RETURNING
*",
 [name, email]
 )
 return result.rows[0]
}
```

RETURNING * returns the inserted row with generated id.

Select (Read):

```
async function getUsers() {
 let result = await pool.query("SELECT * FROM users ORDER BY
created_at DESC")
 return result.rows
}

async function getUserById(id) {
```

```
 let result = await pool.query("SELECT * FROM users WHERE id =
$1", [id])
 return result.rows[0]
}

async function getUserByEmail(email) {
 let result = await pool.query("SELECT * FROM users WHERE
email = $1", [email])
 return result.rows[0]
}
```

Update:

```
async function updateUser(id, name, email) {
 let result = await pool.query(
 "UPDATE users SET name = $1, email = $2 WHERE id = $3
RETURNING *",
 [name, email, id]
 )
 return result.rows[0]
}
```

Delete:

```
async function deleteUser(id) {
 let result = await pool.query("DELETE FROM users WHERE id =
$1 RETURNING *", [id])
 return result.rows[0]
}
```

TRANSACTIONS

Multiple operations that must all succeed or all fail:

```
async function transferMoney(fromId, toId, amount) {
```

```
 let client = await pool.connect()

 try {
 await client.query("BEGIN")

 await client.query(
 "UPDATE accounts SET balance = balance - $1 WHERE id = $2",
 [amount, fromId]
 )

 await client.query(
 "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
 [amount, toId]
 )

 await client.query("COMMIT")
 } catch (error) {
 await client.query("ROLLBACK")
 throw error
 } finally {
 client.release()
 }
}
```

Breaking down:

pool.connect() - Get dedicated connection for transaction
BEGIN - Start transaction
COMMIT - Save all changes
ROLLBACK - Undo all changes if error
client.release() - Return connection to pool

ERROR HANDLING

```
async function createUser(name, email) {
 try {
 let result = await pool.query(
```

```
 "INSERT INTO users (name, email) VALUES ($1, $2) RETURNING
*",
 [name, email]
 )
 return result.rows[0]
 } catch (error) {
 if (error.code === "23505") {
 throw new Error("Email already exists")
 }
 throw error
 }
}
```

PostgreSQL error codes:
- 23505: Unique violation
- 23503: Foreign key violation
- 23502: Not null violation

EXPRESS INTEGRATION

```
const express = require("express")
const { Pool } = require("pg")

let app = express()
app.use(express.json())

let pool = new Pool({
 connectionString: process.env.DATABASE_URL
})

app.get("/api/users", async (req, res, next) => {
 try {
 let result = await pool.query("SELECT id, name, email FROM
users")
 res.json(result.rows)
 } catch (error) {
 next(error)
```

```
 }
})

app.get("/api/users/:id", async (req, res, next) => {
 try {
 let result = await pool.query(
 "SELECT id, name, email FROM users WHERE id = $1",
 [req.params.id]
 )

 if (result.rows.length === 0) {
 return res.status(404).json({ error: "User not found" })
 }

 res.json(result.rows[0])
 } catch (error) {
 next(error)
 }
})

app.post("/api/users", async (req, res, next) => {
 try {
 let { name, email } = req.body
 let result = await pool.query(
 "INSERT INTO users (name, email) VALUES ($1, $2) RETURNING
id, name, email",
 [name, email]
 )
 res.status(201).json(result.rows[0])
 } catch (error) {
 next(error)
 }
})

app.listen(3000)
```

ENVIRONMENT CONFIGURATION

Use environment variables for database credentials:

.env:

DATABASE_URL=postgresql://
myuser:mypassword@localhost:5432/myapp

Code:

```
require("dotenv").config()

let pool = new Pool({
 connectionString: process.env.DATABASE_URL
})
```

Never commit passwords to version control.

PRACTICE

Install PostgreSQL locally.

Create a database and user.

Write CRUD functions with pg.

Build an Express API with PostgreSQL.

Practice transactions.

WHAT YOU LEARNED

You learned PostgreSQL installation.
You learned the pg driver.

You learned parameterized queries.
You learned connection pooling.
You learned CRUD operations.
You learned transactions.
You learned Express integration.

The next chapter teaches MongoDB basics.

BOOK 15: DATABASES

Chapter 4: MongoDB Basics

WHY MONGODB

MongoDB is a document database.

Data stored as JSON-like documents.
No fixed schema required.
Natural fit for JavaScript.

Features:
- Flexible documents
- Horizontal scaling
- Rich query language
- Aggregation pipeline
- Great for rapid development

INSTALLING MONGODB

On Ubuntu:

```
wget -qO - https://www.mongodb.org/static/pgp/server-7.0.asc | sudo apt-key add -
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu jammy/mongodb-org/7.0 multiverse" | sudo tee /etc/apt/
```

sources.list.d/mongodb-org-7.0.list
sudo apt update
sudo apt install mongodb-org
sudo systemctl start mongod
sudo systemctl enable mongod

On macOS:

brew tap mongodb/brew
brew install mongodb-community
brew services start mongodb-community

Check installation:

mongosh

Or use MongoDB Atlas (cloud) for free tier.

MONGODB CONCEPTS

Database:
Container for collections.
Like a PostgreSQL database.

Collection:
Group of documents.
Like a SQL table but no fixed schema.

Document:
Single record as BSON (Binary JSON).
Like a SQL row but flexible.

Example document:

{

"_id": ObjectId("507f1f77bcf86cd799439011"),
"name": "Alice",
"email": "alice@example.com",
"age": 25,
"tags": ["developer", "javascript"],
"address": {
"city": "New York",
"country": "USA"
}
}

Documents can have:
- Nested objects
- Arrays
- Different fields per document

MONGODB SHELL

Connect:

mongosh

Show databases:

show dbs

Use database (creates if not exists):

use myapp

Show collections:

show collections

Create collection implicitly:

db.users.insertOne({ name: "Alice" })

NODE.JS MONGODB DRIVER

Install:

npm install mongodb

Connect:

```
const { MongoClient } = require("mongodb")

let url = "mongodb://localhost:27017"
let client = new MongoClient(url)

async function main() {
 await client.connect()
 console.log("Connected to MongoDB")

 let db = client.db("myapp")
 let users = db.collection("users")

 await client.close()
}

main()
```

Breaking down:

MongoClient - Client for connecting
url - MongoDB connection URL
client.connect() - Establish connection
client.db("myapp") - Get database
db.collection("users") - Get collection

INSERT DOCUMENTS

Insert one:

```
const { MongoClient } = require("mongodb")

let client = new MongoClient("mongodb://localhost:27017")

async function createUser(userData) {
 await client.connect()
 let db = client.db("myapp")
 let users = db.collection("users")

 let result = await users.insertOne({
 name: userData.name,
 email: userData.email,
 createdAt: new Date()
 })

 console.log("Inserted ID:", result.insertedId)
 return result
}
```

result.insertedId is the generated _id.

Insert many:

```
async function createUsers(usersData) {
 let db = client.db("myapp")
 let users = db.collection("users")

 let result = await users.insertMany([
 { name: "Alice", email: "alice@example.com" },
 { name: "Bob", email: "bob@example.com" },
 { name: "Charlie", email: "charlie@example.com" }
 ])

 console.log("Inserted count:", result.insertedCount)
```

```
 return result
}
```

FIND DOCUMENTS

Find all:

```
async function getUsers() {
 let db = client.db("myapp")
 let users = db.collection("users")

 let result = await users.find({}).toArray()
 return result
}
```

find({}) - Empty filter returns all documents.
toArray() - Convert cursor to array.

Find with filter:

```
async function getUsersByAge(minAge) {
 let db = client.db("myapp")
 let users = db.collection("users")

 let result = await users.find({ age: { $gte: minAge } }).toArray()
 return result
}
```

$gte means "greater than or equal".

Find one:

```
async function getUserByEmail(email) {
 let db = client.db("myapp")
 let users = db.collection("users")
```

```
 let user = await users.findOne({ email: email })
 return user
}
```

findOne returns single document or null.

Find by ID:

```
const { ObjectId } = require("mongodb")

async function getUserById(id) {
 let db = client.db("myapp")
 let users = db.collection("users")

 let user = await users.findOne({ _id: new ObjectId(id) })
 return user
}
```

_id must be ObjectId type, not string.

QUERY OPERATORS

Comparison:

$eq - Equal
$ne - Not equal
$gt - Greater than
$gte - Greater than or equal
$lt - Less than
$lte - Less than or equal
$in - In array
$nin - Not in array

```
users.find({ age: { $gte: 18, $lte: 65 } })
```

users.find({ status: { $in: ["active", "pending"] } })

Logical:

$and - All conditions must match
$or - Any condition must match
$not - Negates condition
$nor - None of conditions match

```
users.find({
 $or: [
 { age: { $lt: 18 } },
 { age: { $gt: 65 } }
 ]
})
```

```
users.find({
 $and: [
 { status: "active" },
 { age: { $gte: 18 } }
 ]
})
```

Element:

$exists - Field exists
$type - Field is specific type

users.find({ middleName: { $exists: true } })

Array:

$all - Array contains all values
$elemMatch - Element matches conditions
$size - Array is specific size

users.find({ tags: { $all: ["developer", "javascript"] } })

users.find({ tags: { $size: 3 } })

Text search:

users.createIndex({ name: "text", bio: "text" })

users.find({ $text: { $search: "developer javascript" } })

PROJECTION

Return only specific fields:

```
async function getUserNames() {
 let db = client.db("myapp")
 let users = db.collection("users")

 let result = await users.find({}, {
 projection: { name: 1, email: 1, _id: 0 }
 }).toArray()

 return result
}
```

1 means include field.
0 means exclude field.
_id included by default unless explicitly excluded.

SORTING AND LIMITING

Sort:

users.find({}).sort({ createdAt: -1 }).toArray()

1 for ascending.
-1 for descending.

Limit:

```
users.find({}).limit(10).toArray()
```

Skip (for pagination):

```
users.find({}).skip(20).limit(10).toArray()
```

Combined:

```
async function getUsers(page = 1, limit = 10) {
 let db = client.db("myapp")
 let users = db.collection("users")

 let skip = (page - 1) * limit

 let result = await users
 .find({})
 .sort({ createdAt: -1 })
 .skip(skip)
 .limit(limit)
 .toArray()

 return result
}
```

UPDATE DOCUMENTS

Update one:

```
async function updateUser(id, updates) {
 let db = client.db("myapp")
 let users = db.collection("users")

 let result = await users.updateOne(
 { _id: new ObjectId(id) },
 { $set: { ...updates, updatedAt: new Date() } }
 )

 return result.modifiedCount
}
```

$set updates specific fields without replacing entire document.

Update operators:

$set - Set field value
$unset - Remove field
$inc - Increment number
$push - Add to array
$pull - Remove from array
$addToSet - Add to array if not exists

```
users.updateOne(
 { _id: new ObjectId(id) },
 { $inc: { viewCount: 1 } }
)

users.updateOne(
 { _id: new ObjectId(id) },
 { $push: { tags: "newTag" } }
)
```

Update many:

```
users.updateMany(
 { status: "pending" },
 { $set: { status: "active" } }
```

)

Replace document:

```
users.replaceOne(
 { _id: new ObjectId(id) },
 { name: "New Name", email: "new@example.com" }
)
```

Replaces entire document (except _id).

DELETE DOCUMENTS

Delete one:

```
async function deleteUser(id) {
 let db = client.db("myapp")
 let users = db.collection("users")

 let result = await users.deleteOne({ _id: new ObjectId(id) })
 return result.deletedCount
}
```

Delete many:

```
users.deleteMany({ status: "inactive" })

users.deleteMany({})
```

INDEXES

Create index for faster queries:

users.createIndex({ email: 1 })

users.createIndex({ email: 1 }, { unique: true })

users.createIndex({ name: 1, email: 1 })

users.createIndex({ createdAt: -1 })

List indexes:

users.indexes()

Drop index:

users.dropIndex("email_1")

PRACTICE

Install MongoDB locally or use Atlas.

Practice insert, find, update, delete.

Use query operators.

Build CRUD functions.

Create indexes.

WHAT YOU LEARNED

You learned MongoDB concepts.

You learned the MongoDB driver.
You learned CRUD operations.
You learned query operators.
You learned projection, sorting, limiting.
You learned update operators.
You learned indexes.

The next chapter teaches Mongoose ODM.

BOOK 15: DATABASES

Chapter 5: Mongoose ODM

WHAT IS MONGOOSE

Mongoose is an Object Document Mapper for MongoDB.

It provides:
- Schema definitions
- Data validation
- Type casting
- Query building
- Middleware hooks
- Population (joins)

Mongoose adds structure to MongoDB's flexibility.

INSTALLING MONGOOSE

```
npm install mongoose
```

Connect:

```
const mongoose = require("mongoose")
```

```
mongoose.connect("mongodb://localhost:27017/myapp")
 .then(() => console.log("Connected to MongoDB"))
 .catch(err => console.error("Connection error:", err))
```

Or with options:

```
mongoose.connect("mongodb://localhost:27017/myapp", {
 useNewUrlParser: true,
 useUnifiedTopology: true
})
```

DEFINING SCHEMAS

Schema defines document structure:

```
const mongoose = require("mongoose")

let userSchema = new mongoose.Schema({
 name: String,
 email: String,
 age: Number,
 active: Boolean,
 createdAt: Date
})

let User = mongoose.model("User", userSchema)
```

Breaking down:

Schema - Defines structure
model - Creates model from schema
"User" - Model name (collection will be "users")

SCHEMA TYPES

Basic types:

```
let schema = new mongoose.Schema({
 name: String,
 age: Number,
 active: Boolean,
 birthday: Date,
 data: Buffer,
 mixed: mongoose.Schema.Types.Mixed,
 arrayOfStrings: [String],
 arrayOfNumbers: [Number],
 nestedObject: {
 field1: String,
 field2: Number
 }
})
```

Type with options:

```
let userSchema = new mongoose.Schema({
 name: {
 type: String,
 required: true,
 trim: true,
 minlength: 2,
 maxlength: 100
 },
 email: {
 type: String,
 required: true,
 unique: true,
 lowercase: true,
 match: /^[^\s@]+@[^\s@]+\.[^\s@]+$/
 },
 age: {
 type: Number,
 min: 0,
```

```
 max: 150
 },
 role: {
 type: String,
 enum: ["user", "admin", "moderator"],
 default: "user"
 },
 createdAt: {
 type: Date,
 default: Date.now
 }
})
```

Options:

required - Field must exist
unique - No duplicates
default - Default value
trim - Remove whitespace
lowercase/uppercase - Transform case
minlength/maxlength - String length limits
min/max - Number limits
enum - Allowed values
match - Regex pattern

CREATING DOCUMENTS

Using new and save:

```
let user = new User({
 name: "Alice",
 email: "alice@example.com",
 age: 25
})

await user.save()
console.log("Created:", user._id)
```

Using create:

```
let user = await User.create({
 name: "Bob",
 email: "bob@example.com",
 age: 30
})
```

Create many:

```
let users = await User.insertMany([
 { name: "Alice", email: "alice@example.com" },
 { name: "Bob", email: "bob@example.com" }
])
```

QUERYING DOCUMENTS

Find all:

```
let users = await User.find()
```

Find with filter:

```
let activeUsers = await User.find({ active: true })

let adults = await User.find({ age: { $gte: 18 } })
```

Find one:

```
let user = await User.findOne({ email: "alice@example.com" })
```

Find by ID:

```
let user = await User.findById("507f1f77bcf86cd799439011")
```

Query builder:

```
let users = await User
 .find({ active: true })
 .where("age").gte(18).lte(65)
 .sort({ createdAt: -1 })
 .limit(10)
 .select("name email")
 .exec()
```

Select specific fields:

```
let users = await User.find().select("name email -_id")
```

- excludes field.

Lean queries (plain objects):

```
let users = await User.find().lean()
```

lean() returns plain JavaScript objects instead of Mongoose documents.
Faster but no Mongoose methods.

UPDATING DOCUMENTS

findByIdAndUpdate:

```
let user = await User.findByIdAndUpdate(
 id,
```

```
 { name: "New Name", age: 26 },
 { new: true, runValidators: true }
)
```

new: true - Return updated document.
runValidators: true - Run schema validation.

findOneAndUpdate:

```
let user = await User.findOneAndUpdate(
 { email: "alice@example.com" },
 { $inc: { loginCount: 1 } },
 { new: true }
)
```

updateMany:

```
let result = await User.updateMany(
 { active: false },
 { $set: { deleted: true } }
)

console.log("Modified:", result.modifiedCount)
```

Save method:

```
let user = await User.findById(id)
user.name = "Updated Name"
user.age = 27
await user.save()
```

DELETING DOCUMENTS

findByIdAndDelete:

let user = await User.findByIdAndDelete(id)

findOneAndDelete:

let user = await User.findOneAndDelete({ email:
"alice@example.com" })

deleteMany:

let result = await User.deleteMany({ active: false })
console.log("Deleted:", result.deletedCount)

VALIDATION

Built-in validators:

```
let userSchema = new mongoose.Schema({
 name: {
 type: String,
 required: [true, "Name is required"],
 minlength: [2, "Name must be at least 2 characters"]
 },
 email: {
 type: String,
 required: [true, "Email is required"],
 validate: {
 validator: function(v) {
 return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(v)
 },
 message: "Invalid email format"
 }
 },
 age: {
 type: Number,
```

```
 min: [0, "Age cannot be negative"],
 max: [150, "Age too high"]
 }
})
```

Custom validators:

```
let userSchema = new mongoose.Schema({
 username: {
 type: String,
 validate: {
 validator: async function(v) {
 let count = await User.countDocuments({ username: v })
 return count === 0
 },
 message: "Username already exists"
 }
 }
})
```

Handling validation errors:

```
try {
 await user.save()
} catch (error) {
 if (error.name === "ValidationError") {
 let errors = Object.values(error.errors).map(e => e.message)
 console.log("Validation failed:", errors)
 }
}
```

MIDDLEWARE (HOOKS)

Pre middleware runs before operation:

```
userSchema.pre("save", function(next) {
 this.updatedAt = new Date()
 next()
})

userSchema.pre("save", async function(next) {
 if (this.isModified("password")) {
 this.password = await bcrypt.hash(this.password, 10)
 }
 next()
})
```

Post middleware runs after operation:

```
userSchema.post("save", function(doc) {
 console.log("User saved:", doc._id)
})

userSchema.post("remove", function(doc) {
 console.log("User removed:", doc._id)
})
```

Query middleware:

```
userSchema.pre("find", function() {
 this.where({ deleted: { $ne: true } })
})
```

INSTANCE METHODS

Methods available on document instances:

```
userSchema.methods.getFullName = function() {
 return `${this.firstName} ${this.lastName}`
}
```

```
userSchema.methods.comparePassword = async
function(password) {
 return bcrypt.compare(password, this.password)
}

let User = mongoose.model("User", userSchema)

let user = await User.findById(id)
console.log(user.getFullName())

let isValid = await user.comparePassword("mypassword")
```

STATIC METHODS

Methods available on model:

```
userSchema.statics.findByEmail = function(email) {
 return this.findOne({ email: email.toLowerCase() })
}

userSchema.statics.findActive = function() {
 return this.find({ active: true })
}

let User = mongoose.model("User", userSchema)

let user = await User.findByEmail("alice@example.com")
let activeUsers = await User.findActive()
```

VIRTUALS

Computed properties not stored in database:

```
userSchema.virtual("fullName").get(function() {
 return `${this.firstName} ${this.lastName}`
})

userSchema.virtual("fullName").set(function(name) {
 let parts = name.split(" ")
 this.firstName = parts[0]
 this.lastName = parts[1]
})
```

Include virtuals in JSON:

```
userSchema.set("toJSON", { virtuals: true })
```

COMPLETE MODEL EXAMPLE

```
const mongoose = require("mongoose")
const bcrypt = require("bcrypt")

let userSchema = new mongoose.Schema({
 name: {
 type: String,
 required: [true, "Name is required"],
 trim: true,
 minlength: 2
 },
 email: {
 type: String,
 required: [true, "Email is required"],
 unique: true,
 lowercase: true
 },
 password: {
 type: String,
 required: [true, "Password is required"],
 minlength: 6,
 select: false
```

```
 },
 role: {
 type: String,
 enum: ["user", "admin"],
 default: "user"
 },
 active: {
 type: Boolean,
 default: true
 }
}, {
 timestamps: true
})

userSchema.pre("save", async function(next) {
 if (this.isModified("password")) {
 this.password = await bcrypt.hash(this.password, 10)
 }
 next()
})

userSchema.methods.comparePassword = async
function(password) {
 return bcrypt.compare(password, this.password)
}

userSchema.statics.findByEmail = function(email) {
 return this.findOne({ email }).select("+password")
}

let User = mongoose.model("User", userSchema)

module.exports = User
```

timestamps: true adds createdAt and updatedAt automatically.
select: false hides password by default.

PRACTICE

Create Mongoose models with validation.

Use middleware for password hashing.

Add instance and static methods.

Practice CRUD operations.

Handle validation errors.

WHAT YOU LEARNED

You learned Mongoose schemas.
You learned schema types and validation.
You learned CRUD operations.
You learned middleware hooks.
You learned instance and static methods.
You learned virtuals.

The next chapter teaches database design.
BOOK 15: DATABASES
Chapter 6: Database Design

WHY DESIGN MATTERS

Poor database design causes:
- Slow queries
- Data inconsistency
- Difficult updates
- Wasted storage
- Complex code

Good design prevents these problems.

NORMALIZATION (SQL)

Normalization reduces redundancy.

Unnormalized (bad):

orders table:
id | customer_name | customer_email | product_name | product_price | quantity
1 | Alice | alice@ex.com | Widget | 10.00 | 2
2 | Alice | alice@ex.com | Gadget | 20.00 | 1
3 | Bob | bob@ex.com | Widget | 10.00 | 3

Problems:
- Customer info repeated
- Product info repeated
- Update customer email requires many changes
- Inconsistency risk

First Normal Form (1NF):

Each column has atomic values.
No repeating groups.

Bad:
id | name | phones
1 | Alice | 555-1234, 555-5678

Good:
id | name | phone
1 | Alice | 555-1234
1 | Alice | 555-5678

Or separate phone table.

Second Normal Form (2NF):

All non-key columns depend on entire primary key.

Separate into logical tables:

customers:
id | name | email
1 | Alice | alice@ex.com
2 | Bob | bob@ex.com

products:
id | name | price
1 | Widget | 10.00
2 | Gadget | 20.00

orders:
id | customer_id | product_id | quantity
1 | 1 | 1 | 2
2 | 1 | 2 | 1
3 | 2 | 1 | 3

Third Normal Form (3NF):

No transitive dependencies.
Non-key columns don't depend on other non-key columns.

Bad:
employees:
id | name | department_id | department_name

department_name depends on department_id, not on id.

Good:
employees:
id | name | department_id

departments:
id | name

RELATIONSHIPS

One-to-One:

Each record in A relates to one record in B.

users:
id | name | email

profiles:
id | user_id | bio | avatar

user_id references users.id with UNIQUE constraint.

One-to-Many:

Each record in A relates to many records in B.

users:
id | name | email

posts:
id | user_id | title | content

user_id references users.id.
One user has many posts.

Many-to-Many:

Records in A relate to many in B, and vice versa.
Requires junction table.

users:
id | name

roles:
id | name

user_roles:
user_id | role_id

user_roles connects users and roles.

SQL SCHEMA EXAMPLE

```
CREATE TABLE users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100) NOT NULL,
 email VARCHAR(255) UNIQUE NOT NULL,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE posts (
 id SERIAL PRIMARY KEY,
 user_id INTEGER NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
 title VARCHAR(200) NOT NULL,
 content TEXT,
 published BOOLEAN DEFAULT false,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE tags (
 id SERIAL PRIMARY KEY,
 name VARCHAR(50) UNIQUE NOT NULL
);

CREATE TABLE post_tags (
```

```
 post_id INTEGER REFERENCES posts(id) ON DELETE CASCADE,
 tag_id INTEGER REFERENCES tags(id) ON DELETE CASCADE,
 PRIMARY KEY (post_id, tag_id)
);

CREATE TABLE comments (
 id SERIAL PRIMARY KEY,
 post_id INTEGER NOT NULL REFERENCES posts(id) ON DELETE
CASCADE,
 user_id INTEGER NOT NULL REFERENCES users(id) ON DELETE
CASCADE,
 content TEXT NOT NULL,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

REFERENCES creates foreign key.
ON DELETE CASCADE deletes related records.

MONGODB SCHEMA DESIGN

MongoDB allows different approaches.

Embedding (denormalization):

Store related data in same document.

```
let postSchema = new mongoose.Schema({
 title: String,
 content: String,
 author: {
 name: String,
 email: String
 },
 comments: [{
 user: String,
 content: String,
 createdAt: Date
```

```
 }],
 tags: [String]
})
```

Pros:
- Single query gets all data
- Atomic updates
- Good for data accessed together

Cons:
- Document size limit (16MB)
- Duplicate data
- Hard to query embedded data independently

Referencing (normalization):

Store references to other documents.

```
let userSchema = new mongoose.Schema({
 name: String,
 email: String
})

let postSchema = new mongoose.Schema({
 title: String,
 content: String,
 author: {
 type: mongoose.Schema.Types.ObjectId,
 ref: "User"
 }
})

let commentSchema = new mongoose.Schema({
 post: {
 type: mongoose.Schema.Types.ObjectId,
 ref: "Post"
 },
 user: {
 type: mongoose.Schema.Types.ObjectId,
```

```
 ref: "User"
 },
 content: String
})
```

Pros:
- No duplicate data
- Documents stay small
- Independent updates

Cons:
- Multiple queries needed
- No atomic updates across collections

WHEN TO EMBED VS REFERENCE

Embed when:
- Data is accessed together
- One-to-few relationship
- Data doesn't change often
- Child data doesn't need independent access

Reference when:
- Large amount of related data
- Many-to-many relationships
- Data changes frequently
- Data needs independent access

Hybrid approach:

Embed frequently accessed fields, reference full document.

```
let postSchema = new mongoose.Schema({
 title: String,
 author: {
 _id: mongoose.Schema.Types.ObjectId,
```

```
 name: String
 },
 commentCount: Number
})
```

Store author name for display.
Reference _id for full profile.
Store commentCount to avoid counting.

MONGOOSE POPULATION

Populate references with actual documents:

```
let postSchema = new mongoose.Schema({
 title: String,
 author: {
 type: mongoose.Schema.Types.ObjectId,
 ref: "User"
 }
})

let Post = mongoose.model("Post", postSchema)

let post = await Post.findById(id).populate("author")

console.log(post.author.name)
```

Populate specific fields:

```
let post = await Post.findById(id)
 .populate("author", "name email")
```

Nested population:

```
let post = await Post.findById(id)
```

```
 .populate({
 path: "comments",
 populate: {
 path: "user",
 select: "name"
 }
 })
```

Virtual population:

```
userSchema.virtual("posts", {
 ref: "Post",
 localField: "_id",
 foreignField: "author"
})

let user = await User.findById(id).populate("posts")
```

INDEXES

Indexes speed up queries but slow writes.

Create indexes on:
- Fields used in WHERE/find
- Fields used in ORDER BY/sort
- Foreign keys
- Fields used in JOIN/populate

SQL indexes:

```
CREATE INDEX idx_posts_user_id ON posts(user_id);

CREATE INDEX idx_posts_created ON posts(created_at DESC);

CREATE UNIQUE INDEX idx_users_email ON users(email);
```

```
CREATE INDEX idx_posts_search ON posts(title, content);
```

Mongoose indexes:

```
let postSchema = new mongoose.Schema({
 title: { type: String, index: true },
 author: { type: ObjectId, ref: "User", index: true },
 tags: [{ type: String, index: true }],
 createdAt: { type: Date, index: true }
})

postSchema.index({ title: "text", content: "text" })

postSchema.index({ author: 1, createdAt: -1 })
```

Compound indexes:

For queries using multiple fields together.

```
postSchema.index({ author: 1, published: 1, createdAt: -1 })
```

Order matters. Leftmost fields most important.

DESIGN PATTERNS

Soft delete:

Don't actually delete, mark as deleted.

```
let userSchema = new mongoose.Schema({
 name: String,
 deleted: { type: Boolean, default: false },
 deletedAt: Date
})
```

```
userSchema.pre("find", function() {
 this.where({ deleted: { $ne: true } })
})

async function softDelete(id) {
 await User.findByIdAndUpdate(id, {
 deleted: true,
 deletedAt: new Date()
 })
}
```

Timestamps:

Track creation and modification.

```
let schema = new mongoose.Schema({
 name: String
}, {
 timestamps: true
})
```

Versioning:

Track document versions.

```
let schema = new mongoose.Schema({
 name: String,
 version: { type: Number, default: 1 }
})

schema.pre("save", function(next) {
 if (!this.isNew) {
 this.version++
 }
 next()
})
```

Audit log:

Track all changes.

```
let auditSchema = new mongoose.Schema({
 collection: String,
 documentId: mongoose.Schema.Types.ObjectId,
 action: { type: String, enum: ["create", "update", "delete"] },
 changes: mongoose.Schema.Types.Mixed,
 user: { type: mongoose.Schema.Types.ObjectId, ref: "User" },
 timestamp: { type: Date, default: Date.now }
})
```

PRACTICE

Design a blog database (SQL).

Design a blog database (MongoDB).

Decide embedding vs referencing.

Create appropriate indexes.

Implement soft delete.

WHAT YOU LEARNED

You learned normalization.
You learned relationships.
You learned embedding vs referencing.
You learned population.
You learned index design.
You learned common patterns.

The next chapter teaches CRUD operations in depth.
BOOK 15: DATABASES
Chapter 7: CRUD Operations

WHAT IS CRUD

CRUD stands for:
- Create (INSERT / insertOne)
- Read (SELECT / find)
- Update (UPDATE / updateOne)
- Delete (DELETE / deleteOne)

These four operations cover all data manipulation.

POSTGRESQL CRUD

Setup:

```
const { Pool } = require("pg")

let pool = new Pool({
 connectionString: process.env.DATABASE_URL
})
```

Create:

```
async function createUser(name, email, password) {
 let query = `
 INSERT INTO users (name, email, password)
 VALUES ($1, $2, $3)
 RETURNING id, name, email, created_at
```

```
 `
 let result = await pool.query(query, [name, email, password])
 return result.rows[0]
}
```

Read all with pagination:

```
async function getUsers(page = 1, limit = 10) {
 let offset = (page - 1) * limit

 let countResult = await pool.query("SELECT COUNT(*) FROM
users")
 let total = parseInt(countResult.rows[0].count)

 let query = `
 SELECT id, name, email, created_at
 FROM users
 ORDER BY created_at DESC
 LIMIT $1 OFFSET $2
 `
 let result = await pool.query(query, [limit, offset])

 return {
 data: result.rows,
 pagination: {
 page,
 limit,
 total,
 pages: Math.ceil(total / limit)
 }
 }
}
```

Read one:

```
async function getUserById(id) {
 let query = "SELECT id, name, email, created_at FROM users
WHERE id = $1"
```

```
 let result = await pool.query(query, [id])
 return result.rows[0] || null
}
```

Read with filter:

```
async function searchUsers(searchTerm, filters = {}) {
 let conditions = []
 let values = []
 let paramIndex = 1

 if (searchTerm) {
 conditions.push(`(name ILIKE $${paramIndex} OR email ILIKE $
${paramIndex})`)
 values.push(`%${searchTerm}%`)
 paramIndex++
 }

 if (filters.role) {
 conditions.push(`role = $${paramIndex}`)
 values.push(filters.role)
 paramIndex++
 }

 if (filters.active !== undefined) {
 conditions.push(`active = $${paramIndex}`)
 values.push(filters.active)
 paramIndex++
 }

 let whereClause = conditions.length > 0
 ? `WHERE ${conditions.join(" AND ")}`
 : ""

 let query = `SELECT * FROM users ${whereClause} ORDER BY
created_at DESC`
 let result = await pool.query(query, values)
 return result.rows
}
```

Update:

```
async function updateUser(id, updates) {
 let fields = []
 let values = []
 let paramIndex = 1

 Object.entries(updates).forEach(([key, value]) => {
 fields.push(`${key} = $${paramIndex}`)
 values.push(value)
 paramIndex++
 })

 fields.push(`updated_at = NOW()`)
 values.push(id)

 let query = `
 UPDATE users
 SET ${fields.join(", ")}
 WHERE id = $${paramIndex}
 RETURNING id, name, email, updated_at
 `

 let result = await pool.query(query, values)
 return result.rows[0]
}
```

Delete:

```
async function deleteUser(id) {
 let query = "DELETE FROM users WHERE id = $1 RETURNING
id"
 let result = await pool.query(query, [id])
 return result.rowCount > 0
}
```

MONGODB CRUD

Setup:

```
const mongoose = require("mongoose")

let userSchema = new mongoose.Schema({
 name: { type: String, required: true },
 email: { type: String, required: true, unique: true },
 password: { type: String, required: true, select: false },
 role: { type: String, enum: ["user", "admin"], default: "user" },
 active: { type: Boolean, default: true }
}, { timestamps: true })

let User = mongoose.model("User", userSchema)
```

Create:

```
async function createUser(userData) {
 let user = await User.create(userData)
 return {
 id: user._id,
 name: user.name,
 email: user.email,
 createdAt: user.createdAt
 }
}
```

Read all with pagination:

```
async function getUsers(page = 1, limit = 10) {
 let skip = (page - 1) * limit

 let [users, total] = await Promise.all([
 User.find()
 .select("name email createdAt")
```

```
 .sort({ createdAt: -1 })
 .skip(skip)
 .limit(limit)
 .lean(),
 User.countDocuments()
 ])

 return {
 data: users,
 pagination: {
 page,
 limit,
 total,
 pages: Math.ceil(total / limit)
 }
 }
}
```

Read one:

```
async function getUserById(id) {
 let user = await User.findById(id).select("name email
createdAt").lean()
 return user
}
```

Read with filter:

```
async function searchUsers(searchTerm, filters = {}) {
 let query = {}

 if (searchTerm) {
 query.$or = [
 { name: { $regex: searchTerm, $options: "i" } },
 { email: { $regex: searchTerm, $options: "i" } }
 ]
 }
```

```
 if (filters.role) {
 query.role = filters.role
 }

 if (filters.active !== undefined) {
 query.active = filters.active
 }

 let users = await User.find(query)
 .select("name email role active")
 .sort({ createdAt: -1 })
 .lean()

 return users
}
```

Update:

```
async function updateUser(id, updates) {
 let user = await User.findByIdAndUpdate(
 id,
 { $set: updates },
 { new: true, runValidators: true }
 ).select("name email updatedAt")

 return user
}
```

Delete:

```
async function deleteUser(id) {
 let result = await User.findByIdAndDelete(id)
 return result !== null
}
```

CRUD SERVICE PATTERN

Reusable service class:

```
class CrudService {
 constructor(model) {
 this.model = model
 }

 async create(data) {
 return this.model.create(data)
 }

 async findAll(query = {}, options = {}) {
 let { page = 1, limit = 10, sort = { createdAt: -1 }, select = "" }
= options
 let skip = (page - 1) * limit

 let [data, total] = await Promise.all([
 this.model.find(query)
 .select(select)
 .sort(sort)
 .skip(skip)
 .limit(limit)
 .lean(),
 this.model.countDocuments(query)
 ])

 return {
 data,
 pagination: { page, limit, total, pages: Math.ceil(total / limit) }
 }
 }

 async findById(id, select = "") {
 return this.model.findById(id).select(select).lean()
 }

 async findOne(query, select = "") {
 return this.model.findOne(query).select(select).lean()
```

```
 }

 async update(id, data) {
 return this.model.findByIdAndUpdate(
 id,
 { $set: data },
 { new: true, runValidators: true }
 ).lean()
 }

 async delete(id) {
 let result = await this.model.findByIdAndDelete(id)
 return result !== null
 }
}

let userService = new CrudService(User)

let users = await userService.findAll({ active: true }, { page: 1,
limit: 20 })
let user = await userService.findById(id)
await userService.update(id, { name: "New Name" })
```

EXPRESS CRUD ROUTES

```
const express = require("express")
let router = express.Router()

router.get("/", async (req, res, next) => {
 try {
 let { page = 1, limit = 10, search, role, active } = req.query

 let filters = {}
 if (role) filters.role = role
 if (active !== undefined) filters.active = active === "true"

 let result = search
```

```
 ? await searchUsers(search, filters)
 : await getUsers(parseInt(page), parseInt(limit))

 res.json(result)
 } catch (error) {
 next(error)
 }
})

router.get("/:id", async (req, res, next) => {
 try {
 let user = await getUserById(req.params.id)

 if (!user) {
 return res.status(404).json({ error: "User not found" })
 }

 res.json(user)
 } catch (error) {
 next(error)
 }
})

router.post("/", async (req, res, next) => {
 try {
 let user = await createUser(req.body)
 res.status(201).json(user)
 } catch (error) {
 if (error.code === 11000 || error.code === "23505") {
 return res.status(409).json({ error: "Email already exists" })
 }
 next(error)
 }
})

router.put("/:id", async (req, res, next) => {
 try {
 let user = await updateUser(req.params.id, req.body)

 if (!user) {
```

```
 return res.status(404).json({ error: "User not found" })
 }

 res.json(user)
 } catch (error) {
 next(error)
 }
})

router.delete("/:id", async (req, res, next) => {
 try {
 let deleted = await deleteUser(req.params.id)

 if (!deleted) {
 return res.status(404).json({ error: "User not found" })
 }

 res.status(204).end()
 } catch (error) {
 next(error)
 }
})

module.exports = router
```

BULK OPERATIONS

PostgreSQL bulk insert:

```
async function createUsers(usersData) {
 let values = []
 let placeholders = []
 let paramIndex = 1

 usersData.forEach(user => {
 placeholders.push(`($${paramIndex}, $${paramIndex + 1}, $
${paramIndex + 2})`)
```

```
 values.push(user.name, user.email, user.password)
 paramIndex += 3
 })

 let query = `
 INSERT INTO users (name, email, password)
 VALUES ${placeholders.join(", ")}
 RETURNING id, name, email
 `

 let result = await pool.query(query, values)
 return result.rows
}
```

MongoDB bulk operations:

```
async function bulkUpdateUsers(operations) {
 let bulk = User.collection.initializeUnorderedBulkOp()

 operations.forEach(op => {
 if (op.type === "update") {
 bulk.find({ _id: op.id }).updateOne({ $set: op.data })
 } else if (op.type === "delete") {
 bulk.find({ _id: op.id }).deleteOne()
 }
 })

 let result = await bulk.execute()
 return {
 modified: result.nModified,
 deleted: result.nRemoved
 }
}
```

TRANSACTION EXAMPLE

PostgreSQL:

```
async function transferBalance(fromId, toId, amount) {
 let client = await pool.connect()

 try {
 await client.query("BEGIN")

 let fromResult = await client.query(
 "UPDATE accounts SET balance = balance - $1 WHERE id = $2
AND balance >= $1 RETURNING balance",
 [amount, fromId]
 )

 if (fromResult.rowCount === 0) {
 throw new Error("Insufficient balance")
 }

 await client.query(
 "UPDATE accounts SET balance = balance + $1 WHERE id = $2",
 [amount, toId]
 )

 await client.query("COMMIT")

 return { success: true }
 } catch (error) {
 await client.query("ROLLBACK")
 throw error
 } finally {
 client.release()
 }
}
```

MongoDB (with sessions):

```
async function transferBalance(fromId, toId, amount) {
 let session = await mongoose.startSession()
 session.startTransaction()
```

```
try {
let fromAccount = await Account.findOneAndUpdate(
{ _id: fromId, balance: { $gte: amount } },
{ $inc: { balance: -amount } },
{ session, new: true }
)

if (!fromAccount) {
throw new Error("Insufficient balance")
}

await Account.findByIdAndUpdate(
toId,
{ $inc: { balance: amount } },
{ session }
)

await session.commitTransaction()
return { success: true }
} catch (error) {
await session.abortTransaction()
throw error
} finally {
session.endSession()
}
}
```

PRACTICE

Implement CRUD for users and posts.

Add pagination and filtering.

Create a reusable CRUD service.

Implement bulk operations.

Handle transactions.

WHAT YOU LEARNED

You learned PostgreSQL CRUD.
You learned MongoDB CRUD.
You learned the service pattern.
You learned Express route integration.
You learned bulk operations.
You learned transactions.

The next chapter teaches relationships and joins.

BOOK 15: DATABASES
Chapter 8: Relationships and Joins

WHY RELATIONSHIPS MATTER

Real data is connected:
- Users have posts
- Posts have comments
- Orders have products
- Products have categories

Relationships model these connections.

SQL JOINS

Joins combine rows from multiple tables.

Sample tables:

users:
id | name | email
1 | Alice | alice@ex.com
2 | Bob | bob@ex.com
3 | Charlie | charlie@ex.com

posts:
id | user_id | title
1 | 1 | First Post
2 | 1 | Second Post
3 | 2 | Hello World
4 | NULL | Orphan Post

INNER JOIN:

Returns rows where both tables have matching data.

SELECT users.name, posts.title
FROM users
INNER JOIN posts ON users.id = posts.user_id;

Result:
name | title
Alice | First Post
Alice | Second Post
Bob | Hello World

Charlie excluded (no posts).
Orphan Post excluded (no user).

LEFT JOIN:

Returns all rows from left table, matched rows from right.

SELECT users.name, posts.title
FROM users

LEFT JOIN posts ON users.id = posts.user_id;

Result:
name | title
Alice | First Post
Alice | Second Post
Bob | Hello World
Charlie | NULL

All users included.
Charlie shows NULL for title.

RIGHT JOIN:

Returns all rows from right table, matched rows from left.

SELECT users.name, posts.title
FROM users
RIGHT JOIN posts ON users.id = posts.user_id;

Result:
name | title
Alice | First Post
Alice | Second Post
Bob | Hello World
NULL | Orphan Post

All posts included.
Orphan Post shows NULL for name.

FULL OUTER JOIN:

Returns all rows from both tables.

SELECT users.name, posts.title
FROM users
FULL OUTER JOIN posts ON users.id = posts.user_id;

Result:
name | title
Alice | First Post
Alice | Second Post
Bob | Hello World
Charlie | NULL
NULL | Orphan Post

MULTIPLE JOINS

Join more than two tables:

```
SELECT
 users.name AS author,
 posts.title AS post,
 comments.content AS comment
FROM users
INNER JOIN posts ON users.id = posts.user_id
INNER JOIN comments ON posts.id = comments.post_id;
```

With table aliases:

```
SELECT u.name, p.title, c.content
FROM users u
JOIN posts p ON u.id = p.user_id
JOIN comments c ON p.id = c.post_id
WHERE u.id = 1;
```

AGGREGATE WITH JOINS

Count posts per user:

```
SELECT users.name, COUNT(posts.id) AS post_count
```

```
FROM users
LEFT JOIN posts ON users.id = posts.user_id
GROUP BY users.id, users.name
ORDER BY post_count DESC;
```

LEFT JOIN ensures users with 0 posts are included.

Get latest post per user:

```
SELECT users.name, posts.title, posts.created_at
FROM users
JOIN posts ON users.id = posts.user_id
WHERE posts.created_at = (
 SELECT MAX(created_at)
 FROM posts p
 WHERE p.user_id = users.id
);
```

SUBQUERIES

Query within a query:

Users with more than 5 posts:

```
SELECT name, email
FROM users
WHERE id IN (
 SELECT user_id
 FROM posts
 GROUP BY user_id
 HAVING COUNT(*) > 5
);
```

Correlated subquery:

```
SELECT name,
 (SELECT COUNT(*) FROM posts WHERE posts.user_id = users.id)
AS post_count
FROM users;
```

NODE.JS WITH JOINS

```
const { Pool } = require("pg")
let pool = new Pool({ connectionString:
process.env.DATABASE_URL })

async function getPostsWithAuthors() {
 let query = `
 SELECT
 posts.id,
 posts.title,
 posts.content,
 posts.created_at,
 users.id AS author_id,
 users.name AS author_name,
 users.email AS author_email
 FROM posts
 JOIN users ON posts.user_id = users.id
 ORDER BY posts.created_at DESC
 `

 let result = await pool.query(query)

 return result.rows.map(row => ({
 id: row.id,
 title: row.title,
 content: row.content,
 createdAt: row.created_at,
 author: {
 id: row.author_id,
 name: row.author_name,
 email: row.author_email
```

```
 }
 }))
}

async function getUserWithPosts(userId) {
 let userQuery = "SELECT id, name, email FROM users WHERE id
= $1"
 let postsQuery = `
 SELECT id, title, content, created_at
 FROM posts
 WHERE user_id = $1
 ORDER BY created_at DESC
 `

 let [userResult, postsResult] = await Promise.all([
 pool.query(userQuery, [userId]),
 pool.query(postsQuery, [userId])
 ])

 if (userResult.rows.length === 0) {
 return null
 }

 return {
 ...userResult.rows[0],
 posts: postsResult.rows
 }
}
```

MONGODB RELATIONSHIPS

Reference approach:

```
let userSchema = new mongoose.Schema({
 name: String,
 email: String
})
```

```
let postSchema = new mongoose.Schema({
 title: String,
 content: String,
 author: {
 type: mongoose.Schema.Types.ObjectId,
 ref: "User"
 }
})

let commentSchema = new mongoose.Schema({
 content: String,
 post: {
 type: mongoose.Schema.Types.ObjectId,
 ref: "Post"
 },
 author: {
 type: mongoose.Schema.Types.ObjectId,
 ref: "User"
 }
})

let User = mongoose.model("User", userSchema)
let Post = mongoose.model("Post", postSchema)
let Comment = mongoose.model("Comment", commentSchema)
```

MONGOOSE POPULATE

Basic populate:

```
let posts = await Post.find().populate("author")

posts[0].author.name
```

Select specific fields:

```
let posts = await Post.find()
 .populate("author", "name email")
```

Only includes name and email from author.

Multiple populates:

```
let posts = await Post.find()
 .populate("author", "name")
 .populate("category", "name")
```

Nested populate:

```
let comments = await Comment.find()
 .populate("author", "name")
 .populate({
 path: "post",
 select: "title",
 populate: {
 path: "author",
 select: "name"
 }
 })
```

Populate with conditions:

```
let posts = await Post.find()
 .populate({
 path: "comments",
 match: { approved: true },
 select: "content author",
 options: { sort: { createdAt: -1 }, limit: 5 },
 populate: { path: "author", select: "name" }
 })
```

VIRTUAL POPULATE

Define virtual relationship:

```
userSchema.virtual("posts", {
 ref: "Post",
 localField: "_id",
 foreignField: "author"
})

userSchema.set("toJSON", { virtuals: true })
userSchema.set("toObject", { virtuals: true })

let User = mongoose.model("User", userSchema)
```

Use virtual populate:

```
let user = await User.findById(id)
 .populate({
 path: "posts",
 select: "title createdAt",
 options: { sort: { createdAt: -1 } }
 })

console.log(user.posts)
```

AGGREGATION PIPELINE

Complex queries with $lookup (MongoDB join):

```
let results = await Post.aggregate([
 {
 $lookup: {
 from: "users",
 localField: "author",
```

```
 foreignField: "_id",
 as: "authorData"
 }
 },
 {
 $unwind: "$authorData"
 },
 {
 $project: {
 title: 1,
 content: 1,
 "author.name": "$authorData.name",
 "author.email": "$authorData.email"
 }
 }
])
```

Count related documents:

```
let usersWithPostCount = await User.aggregate([
 {
 $lookup: {
 from: "posts",
 localField: "_id",
 foreignField: "author",
 as: "posts"
 }
 },
 {
 $project: {
 name: 1,
 email: 1,
 postCount: { $size: "$posts" }
 }
 },
 {
 $sort: { postCount: -1 }
 }
])
```

EMBEDDED RELATIONSHIPS

When data is always accessed together:

```
let postSchema = new mongoose.Schema({
 title: String,
 content: String,
 author: {
 _id: mongoose.Schema.Types.ObjectId,
 name: String
 },
 comments: [{
 content: String,
 author: {
 _id: mongoose.Schema.Types.ObjectId,
 name: String
 },
 createdAt: { type: Date, default: Date.now }
 }]
})
```

Adding embedded comment:

```
async function addComment(postId, userId, content) {
 let user = await User.findById(userId).select("name").lean()

 let post = await Post.findByIdAndUpdate(
 postId,
 {
 $push: {
 comments: {
 content,
 author: { _id: user._id, name: user.name },
 createdAt: new Date()
 }
```

```
 }
 },
 { new: true }
 )

 return post
}
```

MANY-TO-MANY RELATIONSHIPS

SQL junction table:

```
CREATE TABLE users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100)
);

CREATE TABLE roles (
 id SERIAL PRIMARY KEY,
 name VARCHAR(50)
);

CREATE TABLE user_roles (
 user_id INTEGER REFERENCES users(id),
 role_id INTEGER REFERENCES roles(id),
 PRIMARY KEY (user_id, role_id)
);

async function getUsersWithRoles() {
 let query = `
 SELECT
 users.id,
 users.name,
 array_agg(roles.name) AS roles
 FROM users
 LEFT JOIN user_roles ON users.id = user_roles.user_id
 LEFT JOIN roles ON user_roles.role_id = roles.id
```

```
 GROUP BY users.id
`
 let result = await pool.query(query)
 return result.rows
}
```

MongoDB approach:

```
let userSchema = new mongoose.Schema({
 name: String,
 roles: [{
 type: mongoose.Schema.Types.ObjectId,
 ref: "Role"
 }]
})

let user = await User.findById(id).populate("roles", "name")
```

PRACTICE

Write SQL joins (INNER, LEFT, RIGHT).

Use Mongoose populate.

Implement virtual relationships.

Build aggregation pipelines.

Handle many-to-many relationships.

WHAT YOU LEARNED

You learned SQL joins.

You learned subqueries.
You learned Mongoose populate.
You learned virtual populate.
You learned aggregation $lookup.
You learned many-to-many patterns.

The next chapter teaches migrations and seeding.

BOOK 15: DATABASES

Chapter 9: Migrations and Seeding

WHY MIGRATIONS

Database schemas change over time:
- Add new tables
- Add new columns
- Modify constraints
- Remove deprecated fields

Migrations track these changes:
- Version control for database schema
- Consistent schema across environments
- Rollback capability
- Team coordination

SQL MIGRATIONS CONCEPT

Migration is two functions:

up - Apply the change
down - Reverse the change

Migration 001_create_users.sql:

```
-- up
CREATE TABLE users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100) NOT NULL,
 email VARCHAR(255) UNIQUE NOT NULL,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- down
DROP TABLE users;
```

Migration 002_add_password.sql:

```
-- up
ALTER TABLE users ADD COLUMN password VARCHAR(255);

-- down
ALTER TABLE users DROP COLUMN password;
```

USING NODE-PG-MIGRATE

Install:

```
npm install node-pg-migrate pg
```

Add script to package.json:

```
{
 "scripts": {
 "migrate:create": "node-pg-migrate create",
 "migrate:up": "node-pg-migrate up",
 "migrate:down": "node-pg-migrate down"
 }
}
```

Create migration:

npm run migrate:create -- create_users

Creates migrations/1234567890123_create-users.js:

```
exports.up = pgm => {
 pgm.createTable("users", {
 id: "id",
 name: { type: "varchar(100)", notNull: true },
 email: { type: "varchar(255)", notNull: true, unique: true },
 password: { type: "varchar(255)" },
 role: { type: "varchar(20)", default: "user" },
 created_at: {
 type: "timestamp",
 notNull: true,
 default: pgm.func("current_timestamp")
 }
 })
}

exports.down = pgm => {
 pgm.dropTable("users")
}
```

Run migrations:

DATABASE_URL=postgresql://user:pass@localhost/myapp npm run migrate:up

Rollback:

DATABASE_URL=postgresql://user:pass@localhost/myapp npm run migrate:down

COMMON MIGRATION OPERATIONS

Add column:

```
exports.up = pgm => {
 pgm.addColumn("users", {
 phone: { type: "varchar(20)" },
 verified: { type: "boolean", default: false }
 })
}

exports.down = pgm => {
 pgm.dropColumn("users", ["phone", "verified"])
}
```

Rename column:

```
exports.up = pgm => {
 pgm.renameColumn("users", "name", "full_name")
}

exports.down = pgm => {
 pgm.renameColumn("users", "full_name", "name")
}
```

Add index:

```
exports.up = pgm => {
 pgm.createIndex("posts", "user_id")
 pgm.createIndex("posts", ["status", "created_at"])
}

exports.down = pgm => {
 pgm.dropIndex("posts", "user_id")
 pgm.dropIndex("posts", ["status", "created_at"])
}
```

Add foreign key:

```
exports.up = pgm => {
 pgm.createTable("posts", {
 id: "id",
 user_id: {
 type: "integer",
 notNull: true,
 references: "users",
 onDelete: "CASCADE"
 },
 title: { type: "varchar(200)", notNull: true },
 content: { type: "text" },
 created_at: {
 type: "timestamp",
 default: pgm.func("current_timestamp")
 }
 })
}

exports.down = pgm => {
 pgm.dropTable("posts")
}
```

Alter column:

```
exports.up = pgm => {
 pgm.alterColumn("users", "name", {
 type: "varchar(200)",
 notNull: true
 })
}

exports.down = pgm => {
 pgm.alterColumn("users", "name", {
 type: "varchar(100)",
 notNull: true
 })
}
```

MONGODB MIGRATIONS

MongoDB schema is flexible, but migrations still useful.

Using migrate-mongo:

npm install migrate-mongo

Initialize:

npx migrate-mongo init

Creates migrate-mongo-config.js:

```
module.exports = {
 mongodb: {
 url: process.env.DATABASE_URL || "mongodb://localhost:27017",
 databaseName: "myapp"
 },
 migrationsDir: "migrations",
 changelogCollectionName: "changelog"
}
```

Create migration:

npx migrate-mongo create add-user-indexes

Creates migrations/20240101120000-add-user-indexes.js:

```
module.exports = {
 async up(db, client) {
 await db.collection("users").createIndex(
 { email: 1 },
 { unique: true }
 )
```

```
 await db.collection("users").createIndex(
 { createdAt: -1 }
 )
 },

 async down(db, client) {
 await db.collection("users").dropIndex("email_1")
 await db.collection("users").dropIndex("createdAt_-1")
 }
}
```

Data migration:

```
module.exports = {
 async up(db, client) {
 await db.collection("users").updateMany(
 { role: { $exists: false } },
 { $set: { role: "user" } }
 )
 },

 async down(db, client) {
 await db.collection("users").updateMany(
 { role: "user" },
 { $unset: { role: "" } }
 )
 }
}
```

Run migrations:

```
npx migrate-mongo up
npx migrate-mongo down
npx migrate-mongo status
```

SEEDING

Seeds populate database with initial or test data.

PostgreSQL seed file:

```
const { Pool } = require("pg")
const bcrypt = require("bcrypt")

let pool = new Pool({ connectionString:
process.env.DATABASE_URL })

async function seed() {
 let password = await bcrypt.hash("password123", 10)

 await pool.query(`
 INSERT INTO users (name, email, password, role) VALUES
 ('Admin User', 'admin@example.com', $1, 'admin'),
 ('Test User', 'user@example.com', $1, 'user')
 ON CONFLICT (email) DO NOTHING
 `, [password])

 console.log("Users seeded")

 await pool.query(`
 INSERT INTO categories (name, slug) VALUES
 ('Technology', 'technology'),
 ('Business', 'business'),
 ('Lifestyle', 'lifestyle')
 ON CONFLICT (slug) DO NOTHING
 `)

 console.log("Categories seeded")

 await pool.end()
}

seed().catch(console.error)
```

MongoDB seed file:

```
const mongoose = require("mongoose")
const bcrypt = require("bcrypt")
const User = require("./models/User")
const Category = require("./models/Category")

async function seed() {
 await mongoose.connect(process.env.DATABASE_URL)

 await User.deleteMany({})
 await Category.deleteMany({})

 let password = await bcrypt.hash("password123", 10)

 await User.create([
 { name: "Admin User", email: "admin@example.com", password,
role: "admin" },
 { name: "Test User", email: "user@example.com", password, role:
"user" }
 ])

 console.log("Users seeded")

 await Category.create([
 { name: "Technology", slug: "technology" },
 { name: "Business", slug: "business" },
 { name: "Lifestyle", slug: "lifestyle" }
 ])

 console.log("Categories seeded")

 await mongoose.disconnect()
}

seed().catch(console.error)
```

FACTORY FUNCTIONS

Generate realistic test data:

```
const { faker } = require("@faker-js/faker")

function createUser(overrides = {}) {
 return {
 name: faker.person.fullName(),
 email: faker.internet.email(),
 password: "password123",
 role: "user",
 createdAt: faker.date.past(),
 ...overrides
 }
}

function createPost(userId, overrides = {}) {
 return {
 userId,
 title: faker.lorem.sentence(),
 content: faker.lorem.paragraphs(3),
 published: faker.datatype.boolean(),
 createdAt: faker.date.past(),
 ...overrides
 }
}

async function seedWithFaker() {
 let users = Array.from({ length: 10 }, () => createUser())

 await User.insertMany(users)

 let posts = []
 for (let user of users) {
 let userPosts = Array.from(
 { length: faker.number.int({ min: 1, max: 5 }) },
 () => createPost(user._id)
 )
 posts.push(...userPosts)
```

```
 }

 await Post.insertMany(posts)

 console.log(`Seeded ${users.length} users and ${posts.length}
posts`)
}
```

ENVIRONMENT-SPECIFIC SEEDS

seeds/index.js:

```
const development = require("./development")
const test = require("./test")
const production = require("./production")

let seeds = {
 development,
 test,
 production
}

async function seed() {
 let env = process.env.NODE_ENV || "development"
 let seeder = seeds[env]

 if (!seeder) {
 console.log(`No seeder for ${env}`)
 return
 }

 await seeder()
 console.log(`Seeded for ${env}`)
}

seed()
```

seeds/development.js:

```
module.exports = async function() {
 await User.deleteMany({})
 await Post.deleteMany({})

}
```

seeds/production.js:

```
module.exports = async function() {
 let adminExists = await User.findOne({ role: "admin" })

 if (!adminExists) {
 await User.create({
 name: "Admin",
 email: process.env.ADMIN_EMAIL,
 password: process.env.ADMIN_PASSWORD,
 role: "admin"
 })
 }
}
```

PACKAGE.JSON SCRIPTS

```
{
 "scripts": {
 "db:migrate": "node-pg-migrate up",
 "db:rollback": "node-pg-migrate down",
 "db:seed": "node seeds/index.js",
 "db:reset": "npm run db:rollback && npm run db:migrate && npm
run db:seed",
 "db:fresh": "dropdb myapp && createdb myapp && npm run
db:migrate && npm run db:seed"
 }
```

}

MIGRATION BEST PRACTICES

1. Never edit existing migrations.
Create new migration to change schema.

2. Test migrations both ways.
Run up, then down, then up again.

3. Keep migrations small.
One logical change per migration.

4. Use transactions when possible.
All or nothing changes.

5. Backup before production migrations.
Always have rollback plan.

6. Run migrations in CI/CD.
Automate deployment process.

PRACTICE

Set up node-pg-migrate or migrate-mongo.

Create migrations for users, posts, comments.

Write seed files with faker.

Practice rolling back and rerunning.

Create environment-specific seeds.

WHAT YOU LEARNED

You learned why migrations matter.
You learned node-pg-migrate.
You learned migrate-mongo.
You learned common migration operations.
You learned seeding strategies.
You learned best practices.

The next chapter brings everything together with practice programs.

BOOK 15: DATABASES
Chapter 10: Practice Database Programs

BLOG API WITH POSTGRESQL

Complete blog API using PostgreSQL:

```
const express = require("express")
const { Pool } = require("pg")
const bcrypt = require("bcrypt")
const jwt = require("jsonwebtoken")
require("dotenv").config()

let app = express()
app.use(express.json())

let pool = new Pool({
 connectionString: process.env.DATABASE_URL
})

async function initDb() {
 await pool.query(`
```

```
 CREATE TABLE IF NOT EXISTS users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100) NOT NULL,
 email VARCHAR(255) UNIQUE NOT NULL,
 password VARCHAR(255) NOT NULL,
 role VARCHAR(20) DEFAULT 'user',
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
 )
 `)

 await pool.query(`
 CREATE TABLE IF NOT EXISTS posts (
 id SERIAL PRIMARY KEY,
 user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
 title VARCHAR(200) NOT NULL,
 content TEXT,
 published BOOLEAN DEFAULT false,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 updated_at TIMESTAMP
 )
 `)

 await pool.query(`
 CREATE TABLE IF NOT EXISTS comments (
 id SERIAL PRIMARY KEY,
 post_id INTEGER REFERENCES posts(id) ON DELETE CASCADE,
 user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
 content TEXT NOT NULL,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
 )
 `)

 await pool.query("CREATE INDEX IF NOT EXISTS idx_posts_user
ON posts(user_id)")
 await pool.query("CREATE INDEX IF NOT EXISTS
idx_comments_post ON comments(post_id)")
}

function authenticate(req, res, next) {
 let token = req.headers.authorization?.replace("Bearer ", "")
```

```
 if (!token) {
 return res.status(401).json({ error: "No token" })
 }

 try {
 req.user = jwt.verify(token, process.env.JWT_SECRET)
 next()
 } catch {
 res.status(401).json({ error: "Invalid token" })
 }
}

app.post("/api/register", async (req, res) => {
 try {
 let { name, email, password } = req.body
 let hash = await bcrypt.hash(password, 10)

 let result = await pool.query(
 "INSERT INTO users (name, email, password) VALUES ($1, $2, $3)
RETURNING id, name, email",
 [name, email, hash]
 )

 let user = result.rows[0]
 let token = jwt.sign({ id: user.id, email: user.email },
process.env.JWT_SECRET, { expiresIn: "7d" })

 res.status(201).json({ user, token })
 } catch (error) {
 if (error.code === "23505") {
 return res.status(409).json({ error: "Email exists" })
 }
 res.status(500).json({ error: error.message })
 }
})

app.post("/api/login", async (req, res) => {
 try {
 let { email, password } = req.body
```

```
 let result = await pool.query("SELECT * FROM users WHERE
email = $1", [email])
 let user = result.rows[0]

 if (!user || !await bcrypt.compare(password, user.password)) {
 return res.status(401).json({ error: "Invalid credentials" })
 }

 let token = jwt.sign({ id: user.id, email: user.email },
process.env.JWT_SECRET, { expiresIn: "7d" })

 res.json({
 user: { id: user.id, name: user.name, email: user.email },
 token
 })
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.get("/api/posts", async (req, res) => {
 try {
 let { page = 1, limit = 10 } = req.query
 let offset = (page - 1) * limit

 let countResult = await pool.query("SELECT COUNT(*) FROM
posts WHERE published = true")
 let total = parseInt(countResult.rows[0].count)

 let result = await pool.query(`
 SELECT p.id, p.title, p.content, p.created_at,
 u.id AS author_id, u.name AS author_name
 FROM posts p
 JOIN users u ON p.user_id = u.id
 WHERE p.published = true
 ORDER BY p.created_at DESC
 LIMIT $1 OFFSET $2
 `, [limit, offset])
```

```
 let posts = result.rows.map(row => ({
 id: row.id,
 title: row.title,
 content: row.content,
 createdAt: row.created_at,
 author: { id: row.author_id, name: row.author_name }
 }))

 res.json({
 data: posts,
 pagination: { page: parseInt(page), limit: parseInt(limit), total }
 })
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.get("/api/posts/:id", async (req, res) => {
 try {
 let postResult = await pool.query(`
 SELECT p.*, u.id AS author_id, u.name AS author_name
 FROM posts p
 JOIN users u ON p.user_id = u.id
 WHERE p.id = $1
 `, [req.params.id])

 if (postResult.rows.length === 0) {
 return res.status(404).json({ error: "Post not found" })
 }

 let commentsResult = await pool.query(`
 SELECT c.*, u.name AS author_name
 FROM comments c
 JOIN users u ON c.user_id = u.id
 WHERE c.post_id = $1
 ORDER BY c.created_at DESC
 `, [req.params.id])

 let row = postResult.rows[0]
 let post = {
```

```
 id: row.id,
 title: row.title,
 content: row.content,
 published: row.published,
 createdAt: row.created_at,
 author: { id: row.author_id, name: row.author_name },
 comments: commentsResult.rows.map(c => ({
 id: c.id,
 content: c.content,
 createdAt: c.created_at,
 author: { id: c.user_id, name: c.author_name }
 }))
 }

 res.json(post)
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.post("/api/posts", authenticate, async (req, res) => {
 try {
 let { title, content, published = false } = req.body

 let result = await pool.query(
 "INSERT INTO posts (user_id, title, content, published) VALUES
($1, $2, $3, $4) RETURNING *",
 [req.user.id, title, content, published]
 )

 res.status(201).json(result.rows[0])
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.put("/api/posts/:id", authenticate, async (req, res) => {
 try {
 let checkResult = await pool.query(
 "SELECT user_id FROM posts WHERE id = $1",
```

```
 [req.params.id]
 )

 if (checkResult.rows.length === 0) {
 return res.status(404).json({ error: "Post not found" })
 }

 if (checkResult.rows[0].user_id !== req.user.id) {
 return res.status(403).json({ error: "Forbidden" })
 }

 let { title, content, published } = req.body

 let result = await pool.query(
 "UPDATE posts SET title = $1, content = $2, published = $3,
updated_at = NOW() WHERE id = $4 RETURNING *",
 [title, content, published, req.params.id]
 )

 res.json(result.rows[0])
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.post("/api/posts/:id/comments", authenticate, async (req, res)
 => {
 try {
 let result = await pool.query(
 "INSERT INTO comments (post_id, user_id, content) VALUES ($1,
$2, $3) RETURNING *",
 [req.params.id, req.user.id, req.body.content]
 )

 res.status(201).json(result.rows[0])
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})
```

```
initDb().then(() => {
 app.listen(3000, () => console.log("Blog API running on port
3000"))
})
```

E-COMMERCE API WITH MONGODB

Complete e-commerce API using MongoDB and Mongoose:

```
const express = require("express")
const mongoose = require("mongoose")
const bcrypt = require("bcrypt")
const jwt = require("jsonwebtoken")
require("dotenv").config()

let app = express()
app.use(express.json())

mongoose.connect(process.env.MONGODB_URL)

let userSchema = new mongoose.Schema({
 name: { type: String, required: true },
 email: { type: String, required: true, unique: true, lowercase: true },
 password: { type: String, required: true, select: false },
 role: { type: String, enum: ["user", "admin"], default: "user" }
}, { timestamps: true })

userSchema.pre("save", async function(next) {
 if (this.isModified("password")) {
 this.password = await bcrypt.hash(this.password, 10)
 }
 next()
})

let productSchema = new mongoose.Schema({
 name: { type: String, required: true },
 description: String,
```

```
 price: { type: Number, required: true, min: 0 },
 category: { type: String, required: true },
 stock: { type: Number, default: 0 },
 images: [String],
 active: { type: Boolean, default: true }
}, { timestamps: true })

productSchema.index({ name: "text", description: "text" })
productSchema.index({ category: 1, price: 1 })

let orderSchema = new mongoose.Schema({
 user: { type: mongoose.Schema.Types.ObjectId, ref: "User",
required: true },
 items: [{
 product: { type: mongoose.Schema.Types.ObjectId, ref: "Product" },
 name: String,
 price: Number,
 quantity: { type: Number, min: 1 }
 }],
 total: { type: Number, required: true },
 status: {
 type: String,
 enum: ["pending", "processing", "shipped", "delivered", "cancelled"],
 default: "pending"
 },
 shippingAddress: {
 street: String,
 city: String,
 country: String,
 postalCode: String
 }
}, { timestamps: true })

let User = mongoose.model("User", userSchema)
let Product = mongoose.model("Product", productSchema)
let Order = mongoose.model("Order", orderSchema)

function authenticate(req, res, next) {
 let token = req.headers.authorization?.replace("Bearer ", "")
```

```
 if (!token) {
 return res.status(401).json({ error: "No token" })
 }

 try {
 req.user = jwt.verify(token, process.env.JWT_SECRET)
 next()
 } catch {
 res.status(401).json({ error: "Invalid token" })
 }
}

function requireAdmin(req, res, next) {
 if (req.user.role !== "admin") {
 return res.status(403).json({ error: "Admin required" })
 }
 next()
}

app.post("/api/register", async (req, res) => {
 try {
 let user = await User.create(req.body)
 let token = jwt.sign(
 { id: user._id, email: user.email, role: user.role },
 process.env.JWT_SECRET,
 { expiresIn: "7d" }
 )

 res.status(201).json({
 user: { id: user._id, name: user.name, email: user.email },
 token
 })
 } catch (error) {
 if (error.code === 11000) {
 return res.status(409).json({ error: "Email exists" })
 }
 res.status(500).json({ error: error.message })
 }
})
```

```
app.post("/api/login", async (req, res) => {
 try {
 let user = await User.findOne({ email: req.body.email
}).select("+password")

 if (!user || !await bcrypt.compare(req.body.password,
user.password)) {
 return res.status(401).json({ error: "Invalid credentials" })
 }

 let token = jwt.sign(
 { id: user._id, email: user.email, role: user.role },
 process.env.JWT_SECRET,
 { expiresIn: "7d" }
 )

 res.json({
 user: { id: user._id, name: user.name, email: user.email },
 token
 })
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.get("/api/products", async (req, res) => {
 try {
 let { page = 1, limit = 12, category, search, sort, minPrice,
maxPrice } = req.query
 let query = { active: true }

 if (category) query.category = category
 if (minPrice || maxPrice) {
 query.price = {}
 if (minPrice) query.price.$gte = parseFloat(minPrice)
 if (maxPrice) query.price.$lte = parseFloat(maxPrice)
 }
 if (search) {
 query.$text = { $search: search }
 }
```

```
 let sortOption = { createdAt: -1 }
 if (sort === "price_asc") sortOption = { price: 1 }
 if (sort === "price_desc") sortOption = { price: -1 }
 if (sort === "name") sortOption = { name: 1 }

 let skip = (page - 1) * limit

 let [products, total] = await Promise.all([
 Product.find(query)
 .sort(sortOption)
 .skip(skip)
 .limit(parseInt(limit))
 .lean(),
 Product.countDocuments(query)
 ])

 res.json({
 data: products,
 pagination: {
 page: parseInt(page),
 limit: parseInt(limit),
 total,
 pages: Math.ceil(total / limit)
 }
 })
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.get("/api/products/:id", async (req, res) => {
 try {
 let product = await Product.findById(req.params.id).lean()

 if (!product) {
 return res.status(404).json({ error: "Product not found" })
 }

 res.json(product)
```

```
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.post("/api/products", authenticate, requireAdmin, async (req,
res) = > {
 try {
 let product = await Product.create(req.body)
 res.status(201).json(product)
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.put("/api/products/:id", authenticate, requireAdmin, async
(req, res) = > {
 try {
 let product = await Product.findByIdAndUpdate(
 req.params.id,
 { $set: req.body },
 { new: true, runValidators: true }
 )

 if (!product) {
 return res.status(404).json({ error: "Product not found" })
 }

 res.json(product)
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.post("/api/orders", authenticate, async (req, res) = > {
 let session = await mongoose.startSession()
 session.startTransaction()

 try {
 let { items, shippingAddress } = req.body
```

```
let orderItems = []
let total = 0

for (let item of items) {
let product = await Product.findOneAndUpdate(
{ _id: item.productId, stock: { $gte: item.quantity } },
{ $inc: { stock: -item.quantity } },
{ session, new: true }
)

if (!product) {
throw new Error(`Product ${item.productId} unavailable or
insufficient stock`)
}

orderItems.push({
product: product._id,
name: product.name,
price: product.price,
quantity: item.quantity
})

total += product.price * item.quantity
}

let order = await Order.create([{
user: req.user.id,
items: orderItems,
total,
shippingAddress
}], { session })

await session.commitTransaction()

res.status(201).json(order[0])
} catch (error) {
await session.abortTransaction()
res.status(400).json({ error: error.message })
} finally {
session.endSession()
```

```
 }
})

app.get("/api/orders", authenticate, async (req, res) => {
 try {
 let query = req.user.role === "admin" ? {} : { user: req.user.id }

 let orders = await Order.find(query)
 .populate("user", "name email")
 .sort({ createdAt: -1 })
 .lean()

 res.json(orders)
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.get("/api/orders/:id", authenticate, async (req, res) => {
 try {
 let order = await Order.findById(req.params.id)
 .populate("user", "name email")
 .lean()

 if (!order) {
 return res.status(404).json({ error: "Order not found" })
 }

 if (req.user.role !== "admin" && order.user._id.toString() !==
req.user.id) {
 return res.status(403).json({ error: "Forbidden" })
 }

 res.json(order)
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.patch("/api/orders/:id/status", authenticate, requireAdmin,
```

```
async (req, res) => {
 try {
 let order = await Order.findByIdAndUpdate(
 req.params.id,
 { status: req.body.status },
 { new: true }
 )

 if (!order) {
 return res.status(404).json({ error: "Order not found" })
 }

 res.json(order)
 } catch (error) {
 res.status(500).json({ error: error.message })
 }
})

app.listen(3000, () => console.log("E-commerce API running on
port 3000"))
```

DATABASE UTILITY MODULE

Reusable database utilities:

```
const mongoose = require("mongoose")
const { Pool } = require("pg")

class DatabaseManager {
 constructor(config) {
 this.config = config
 this.type = config.type
 }

 async connect() {
 if (this.type === "mongodb") {
 await mongoose.connect(this.config.url)
```

```
 console.log("Connected to MongoDB")
 } else if (this.type === "postgresql") {
 this.pool = new Pool({ connectionString: this.config.url })
 await this.pool.query("SELECT NOW()")
 console.log("Connected to PostgreSQL")
 }
 }

 async disconnect() {
 if (this.type === "mongodb") {
 await mongoose.disconnect()
 } else if (this.type === "postgresql") {
 await this.pool.end()
 }
 }

 async healthCheck() {
 try {
 if (this.type === "mongodb") {
 await mongoose.connection.db.admin().ping()
 } else if (this.type === "postgresql") {
 await this.pool.query("SELECT 1")
 }
 return { status: "healthy", type: this.type }
 } catch (error) {
 return { status: "unhealthy", type: this.type, error: error.message }
 }
 }
}

class QueryBuilder {
 constructor(model) {
 this.model = model
 this._query = {}
 this._sort = {}
 this._select = ""
 this._populate = []
 this._skip = 0
 this._limit = 0
 }
```

```
where(conditions) {
this._query = { ...this._query, ...conditions }
return this
}

sort(field, order = "asc") {
this._sort[field] = order === "desc" ? -1 : 1
return this
}

select(fields) {
this._select = fields
return this
}

populate(path, select) {
this._populate.push({ path, select })
return this
}

paginate(page, limit) {
this._skip = (page - 1) * limit
this._limit = limit
return this
}

async execute() {
let query = this.model.find(this._query)

if (Object.keys(this._sort).length > 0) {
query = query.sort(this._sort)
}

if (this._select) {
query = query.select(this._select)
}

for (let pop of this._populate) {
query = query.populate(pop.path, pop.select)
```

```
  }

  if (this._skip > 0) {
  query = query.skip(this._skip)
  }

  if (this._limit > 0) {
  query = query.limit(this._limit)
  }

  return query.lean()
  }

  async count() {
  return this.model.countDocuments(this._query)
  }

  async paginated(page, limit) {
  this.paginate(page, limit)

  let [data, total] = await Promise.all([
  this.execute(),
  this.count()
  ])

  return {
  data,
  pagination: {
  page,
  limit,
  total,
  pages: Math.ceil(total / limit)
  }
  }
  }
}

module.exports = { DatabaseManager, QueryBuilder }
```

WHAT YOU BUILT

You built two complete database-backed applications:

Blog API with PostgreSQL:
- User authentication
- Posts with authors
- Comments system
- SQL joins

E-commerce API with MongoDB:
- Product catalog
- Order processing
- Transaction handling
- Stock management

Database utility module:
- Connection management
- Health checks
- Query builder

WHAT COMES NEXT

You now understand databases:

- Database concepts (Book 15, Chapter 1)
- SQL fundamentals (Book 15, Chapter 2)
- PostgreSQL with Node.js (Book 15, Chapter 3)
- MongoDB basics (Book 15, Chapter 4)
- Mongoose ODM (Book 15, Chapter 5)
- Database design (Book 15, Chapter 6)
- CRUD operations (Book 15, Chapter 7)
- Relationships and joins (Book 15, Chapter 8)
- Migrations and seeding (Book 15, Chapter 9)

Book 16 teaches you React. You will:

- Build user interfaces
- Understand components
- Manage state
- Handle events
- Create single-page applications

You finished Book 15. You persist data. Now you build interfaces.